At the ballot box

Assignment written by Julie Zelenski

In 1887, French mathematician Joseph Bertrand pondered an idealized election conducted between two candidates, **A** and **B**. Each voter casts a ballot for a single candidate and adds it to a sealed box. When the polls close, the box is shaken to mix up the order and ballots are removed one by one and tallied. At the end of the tally, candidate **A** has more votes than **B** and wins the election. Bertrand's question was: What is the likelihood that the winning candidate **A** is strictly ahead of candidate **B** throughout the entire tally?

Although Bertrand posed the question in terms of probability, his answer uses an argument based on counting. **An ordering of the ballots where the eventual winner is always in the lead is considered a *favorable* ordering**. If you count the number of favorable orderings and divide by the count of all possible orderings, this ratio is the likelihood of a random ordering being a favorable one.

Consider an election with only three ballots: two cast for **A** and one for **B**. The ballots can be tallied in three possible orders: **AAB**, **ABA**, and **BAA**:

- **AAB**: **A** takes the lead as the first ballot is tallied and never loses it (**favorable**)
- **ABA**: **A** has an early lead, briefly loses lead before regaining it (**not favorable**)
- **BAA**: **A** gains the lead only when tallying the last ballot (**not favoriable**)

Given that all three orderings are equally likely and only one is favorable, the likelihood of a favorable ordering is 1/3.

Recursion can be a useful technique for counting tasks such as this one!

Count all orderings

Write the recursive function

```
int countAllOrderings(int a, int b)
```

that returns the number of possible orderings for tallying an election with ballot counts **a** and **b**.

Here are some hints on recursively breaking down the task:

- The simplest case is when all remaining ballots are for a single candidate. There is only one possible ordering of the remaining ballots, you will be tallying all for **A** (or **B**). This sounds like a base case!
- Otherwise both candidates have ballots remaining, meaning there are two possibilities for the next ballot to be tallied: either it will be a ballot for **A** or one for **B**. Your counting will consider both.
- Tallying a single ballot leaves you with a slightly simpler form of the same problem to solve. Self-similarity is the hallmark of a recursive case. You will make recursive call(s) to count the orderings for the smaller set of remaining ballots and use those results to solve the original problem.
- In lecture, we considered the use of recursion to generate/print all possible sequences from flipping a coin or rolling a die. Such examples have a similar structure to what is needed here, but a notable difference is that for the problem you do not construct the actual sequences, only count them, which simplifies the task.
- The point of this exercise is to solve the problem recursively. So even if you know another way to derive the result, put that aside, and focus on **recursive counting**. There should be no loops, no multiplication, no factorial, no combinatorics, no binomials, no generating functions, … No fancy math whatsoever, just addition. Also do not use additional data structures (no `Stack`s, `Queue`s, `Vector`s, strings, etc.). Remember that you do not need to generate, print, or store the orderings, just count them.

Testing

- Choose a few small values of **a** and **b** and list out the orderings by hand. Add `STUDENT_TEST` cases to confirm the results from `countAllOrderings` matches your hand counts. Manually constructing these test cases can be tedious, we recommend setting one of **a** or **b** to be a very small number to simplify. -The count of orderings climbs very quickly – a recursive exploration of orderings/permutations exhibits factorial growth `O(N!)`, ouch!

Even for small-ish values of **a** and **b**, the function will begin to slow down and the count will quickly exceed what can be stored in an **`int`**. Manually listing the orderings to count also becomes entirely impractical. In the next section, we offer a clever alternative for broad test coverage without having to hand-verify.

- Your function can assume that **a** and **b** are both non-negative. It is valid for either or both of **a** and **b** to be zero. In light of system limitations, you may also assume that the sum **`a + b`** is <= 30. We will not test your function on arguments that violate these assumptions.

Count favorable orderings

Write a second recursive function

```
int countFavorableOrderings(int a, int b)
```

that returns the number of favorable orderings for tallying an election with ballot counts **a** and **b**. A favorable ordering is one in which candidate **`A`** was in the lead throughout.

Some hints on how to approach:

- The general idea is to copy/paste your **`countAllOrderings`** function and modify it to restrict exploration to only those orderings that meet the criteria for being *favorable*.
- Add an additional argument to the recursive calls to keep track of the current value of **`A`**'s' lead. As long as the value of lead is positive, the current ordering is favorable and you keep counting. Conversely, as soon the lead becomes zero or negative, you'll know there are no orderings to be counted on this path.
- Because of the additional argument needed by the recursive call, you will need to use a wrapper function to set up the first call to the inner recursive function.
- The stopping conditions and handling of base case(s) requires some careful thought. We recommend tracing some simple cases on paper or in the debugger to confirm your understanding of the proper handling.

Testing

Review the orderings you manually listed when constructing test cases for **`countAllOrderings`**. Identify which of those orderings meet the criteria to be favorable. Construct student test cases that confirm the results from **`countFavorableOrderings`** matches your hand counts. As before, you will want to restrict yourself to relatively small values of **a** and **b**.

Bertrand's Ballot Theorem

The wondrous *pièce de résistance* of this story is that Bertrand was able to derive a closed form for the likelihood of a favorable ordering for all values of **a** and **b** where **a** > **b**:

```
                                        (a - b)
Likelihood of favorable ordering   =   --------
                                        (a + b)
```

Such a neat and surprising 🤯 result! You do not need to derive his proof, you may take the formula on faith. (But those of you in CS103 and/or CS109 or just curious should check out the references to learn more!)

Bertrand's nifty theorem gives a new way to confirm the correctness of your functions. Above you had to hand-compute the result to know what value to expect for a test case, which is tedious, error-prone, and doesn't scale. Now that you know another way to compute the likelihood of a favorable ordering, you can write test cases that compare the two results to confirm. Neat!

Write a final **`STUDENT_TEST`** that uses a for loop over a range of values for **a** and **b**, confirming that the ratio of your counts (**`likelyFavorable`**) is equal to the result of Bertrand (**`bertrandTheorem`**). Because of system limitations, you will still need to **restrict your testing to values of `a` and `b` that sum to less than 30**. These smaller values will permit the counting to complete in a reasonable amount of time and will not exceed the range of the **`int`** type. Having confirmed correctness on all of these small values, the power of induction (which is just recursion in disguise) allows you to conclude they also would work correctly on ever larger values, if only our system had the necessary time/memory to permit it.

References

This exercise was inspired by an intriguing Stanford news article about the need for more honest and robust predictive models when reporting election returns.

- https://news.stanford.edu/2021/03/19/honesty-statistical-models/ "A problem with raw tallies of the presidential election is that they create a false narrative that the final results are still developing in drastic ways. In reality, on election night there is no “catching up from behind” or “losing the lead” because the votes are already cast; the winner has already won – we just don’t know it yet. More than being merely imprecise, these riveting descriptions of the voting process can make the outcomes seem excessively suspicious or surprising."

You don't need to delve into the mathematics behind Bertrand's theorem, but it is quite fascinating if you are curious to learn more:

- Wikipedia page on Bertrand's Ballot Theorem
- Mark Renault, *Four Proofs of the Ballot Theorem*